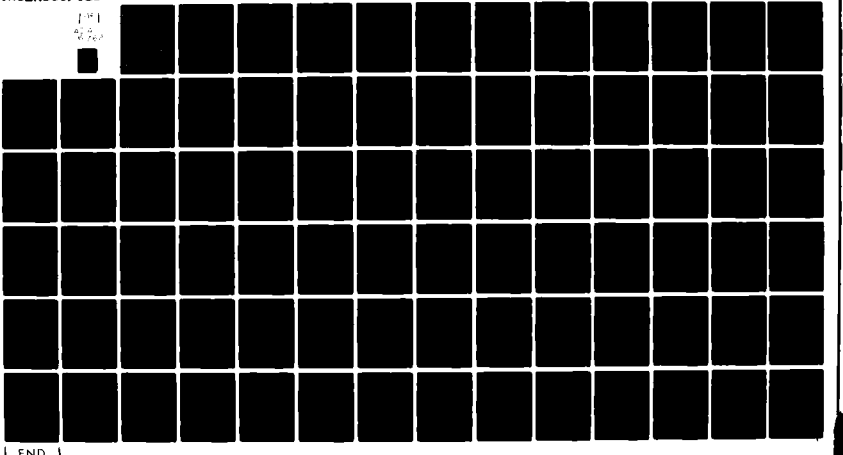
AD-A116 762 CONNECTICUT UNIV STORRS LAB FOR COMPUTER SCIENCE RE--ETC F/G 9/2
MODELS AND MEASUREMENTS OF PARALLELISM FOR A DISTRIBUTED COMPUT--ETC(U)
1982 D S LANE DAS060-79-C-0117

UNCLASSIFIED TR-CS-82-1 NL



END
DATE FILMED
8-82
DTIC

(2)

COMPUTER SCIENCE TECHNICAL REPORT

Laboratory for Computer Science Research
The University of Connecticut

COMPUTER SCIENCE DIVISION

Electrical Engineering and Computer Science Department
U-157
The University of Connecticut
Storrs, Connecticut 06268

(2)

MODELS AND MEASUREMENTS OF PARALLELISM FOR A DISTRIBUTED COMPUTER SYSTEM

Debra S. Lane

Technical Report TR CS-82-1

Contract DAAG 29-81-K-0117

MODELS AND MEASUREMENTS OF PARALLELISM FOR A DISTRIBUTED COMPUTER SYSTEM

Debra S. Lane

B.A., SUNY at Potsdam

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

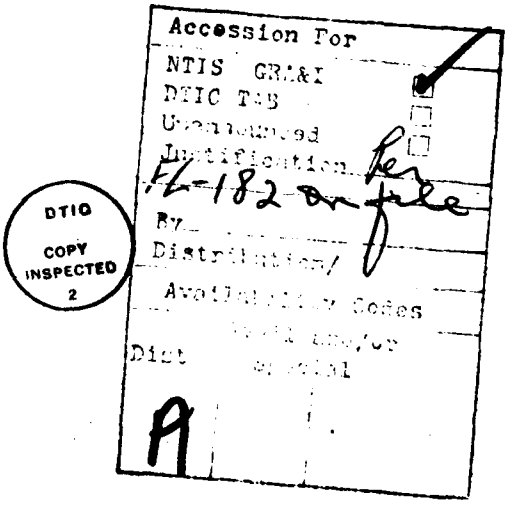
Master of Science

at

The University of Connecticut

1982

TABLE OF CONTENTS



LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Distributed computer systems may be defined as local computer networks executing decentralized software. The decentralization of software is accomplished by logically distributing the control and information structures among the computers. The programming language concepts necessary to decentralize software is currently a topic of research. One approach to constructing such software is through the use of autonomous processes [4,14,20]. Synchronization and communication between processes is achieved by the transmission of unbuffered messages. A process model, along with the ability to send and receive messages in a non-deterministic manner are the most important programming concepts necessary to distribute software.

Another area of research is the implementation of such programming concepts [15,19,26]. The issues here are concerned with the representation of state information in the operating system. There are three possibilities: (1) partitioned state information, (2) replicated state information, or (3) some combination of (1) and (2). The implementation of routines which use and manipulate the state information in the operating system depend upon which representation is chosen. For exam-

ple, if a partitioned implementation is chosen, the algorithms must make decisions based on a partial view of the system state, and/or untimely state information. On the other hand, in the replicated case, the issue is one of maintaining consistency. In any case, the control structure is decentralized.

The design of algorithms with decentralized control structures is yet another research topic in the area of distributed computer systems [7,19]. At this time, the primary advantages of distributed computer systems are increased reliability over centralized systems, and the potential for incremental system growth with a minimum amount of software and hardware redesign. Therefore, decentralized algorithms are not only concerned with implementing operating system routines in a decentralized fashion, but also with continued execution in the case of hardware failures, and the incorporation of new nodes in the system. Currently, the problems being looked at are those of implementing mutual exclusion and maintaining consistency of distributed databases or state information [6,10,17,18].

Once the concepts needed to decentralize software and the associated implementation issues are defined, then the performance of distributed computer systems can be addressed. The parallelism inherent in distributed software suggests that it might outperform a centralized

solution, in the sense that the same amount of work could be done in a shorter time through parallel processing. In order to begin to compare decentralized implementations of software to centralized versions, a first step is to identify the features of software affecting parallelism. Another important reason to analyze the performance of distributed systems is to be able to design distributed systems to perform optimally. In assessing the overall performance of distributed systems, the structure and implementation of the communication medium is seen to be an important factor. It is the software, however, that will determine the communication workload. Knowledge of the behavior and characteristics of the software is important not only in realizing the communication needs, but also in being able to begin to provide guidelines for the design of decentralized programs. The key features of decentralized software are its potential for parallel execution and the delays introduced by communication. It is necessary to understand the tradeoff between these two features in examining the performance of distributed software.

The formulation of a model which explains the parallelism exhibited by an experimental distributed computer system, XDCS, existing at the University of Connecticut is one goal of this thesis. The XDCS system runs an operating system kernel whose implementation is

based on partitioned state information [13]. This kernel performs process and memory management to support the execution of benchmark decentralized algorithms written in the distributed programming language EPL [21,22]. In order to be sure that the features of the system affecting parallelism have been correctly identified and represented, the model is validated.

The other goal of this thesis is the instrumentation of the XDCS system. Instrumenting a system is done in support of an evaluation study intended to answer questions about the performance of the system [12]. The performance information required by a study may be obtained from the system itself (measurement techniques) or from a model of the system (modeling techniques). A problem common to all modeling techniques is one of model accuracy; verification of the model should be performed before it is used to produce the information needed by an evaluation study. The only way to validate the model is through experimentation. Thus, the primary focus of instrumenting the XDCS system is to provide a tool through which the model mentioned above can be validated.

The endeavors of this thesis fall into two well known areas of performance analysis, analytic modeling of computer systems, and measuring computer systems. Although these two areas are related, the issues in-

volved in carrying out these two tasks are not related. The concerns of modeling are those of abstracting the features of a system which influence the performance indices under study. A representation of the features must then be proposed. This representation is the model. The design of measurements entertains the following problems: (1) enough experiments must be performed so that the moments and if possible, the distribution of the quantities of interest can be estimated with sufficient accurracy, (2) the measurements should not interfere with the behavior of the system, or, at least the interference should be estimated and accounted for, (3) restrictions on instrumentation imposed by the system must be considered. Due to the fact that the issues under consideration in modeling and measuring are not the same, these tasks have been separated into two distinct and equally important areas of effort in this thesis.

The discussion of the thesis is organized in the following way: Chapter 2 supplies the necessary background; Chapter 3 discusses those features of the XDCS system that are relevant to the modeling and measuring tasks; a discussion of the model is contained in Chapter 4; the instrumentation of the system is presented in Chapter 5; Chapter 6 contains the results of the exercise and information that can be obtained from the

model; and finally, Chapter 7 summarizes the work and presents the final conclusions. After validating the model, it was discovered that the model was not very accurate. Chapter 6 presents these results and discusses the reasons for them. Even though the model was not as accurate as desired, insight and information about the features of distributed computer systems has been obtained from this study.

CHAPTER 2

BACKGROUND

This work is part of a research project at the University of Connecticut. One goal of the project is to investigate implementations of the programming language concepts necessary for decentralizing application software. Another important part of the research focuses on the performance characteristics of the implementations, and in the process of studying the implementations attempts to isolate and understand the relationship between features of and performance behavior exhibited by distributed computer systems.

Some preliminary performance data obtained through simulation indicated that the cost of decentralization of software might not be as high as was expected [3]. Although it was found that the operating system kernel providing runtime support for the application software was used extensively, potentially producing a large overhead, performance improvements due to the inherent parallelism seemed to offset the high overhead costs. This initial data suggests that an objective of performance models for decentralized software is an explanation of the trade-off between parallelism and the overhead incurred in implementing the parallelism. The measurements for this study were obtained from a single

CPU implementation of the kernel.

A multiple CPU kernel to run the same application software was designed and implemented [13]. Measurements were taken of this kernel to determine the actual overhead costs and to gather statistics about the message traffic. It was expected that the costs incurred would be greater than those obtained from the simulation due to the use of a communication medium which introduces delays in the transmission of messages, and decentralization of the kernel which requires low-level communications to coordinate activities between instances of the kernel. Higher costs were indeed the case. The number of instructions executed for each kernel primitive were five to ten times larger in the multiple CPU implementation than in the single CPU one. Also, much more communication took place and at a higher cost per message. Thus, the overhead costs of distributing hardware, control, and data are substantial.

It is hoped that parallel execution of the processes comprising an application program will defray the overhead costs of distributed computing. This parallelism needs to be quantified and explained in terms of the relevent system features.

2.1 STUDIES OF PARALLELISM

Multiprogramming, the execution of more than one program at the same time on a single processor, made possible by the use of synchronization and virtual memory techniques, is a form of parallel processing. With the advent of multiprocessors, parallel processing assumed another definition. Multiprocessor architectures consist of multiple processors which share primary memory. This type of architecture makes it possible to execute parts of one program simultaneously or to execute many programs at the same time. The next type of architecture to emerge has been called distributed. The definition of a distributed architecture is broader than and actually encompasses some multiprocessor systems. A distributed computer system architecture consists of a collection of processors connected in some fashion so that communication between processors is possible. Parallel processing in distributed computer systems is the topic of interest here.

The first models or studies of parallelism ocurred when multiprocessor architectures appeared. The goal of these models was initially to represent parallel programs and/or the possible parallel execution of portions of a program [2]. Next these models were used and expanded, or new representations were proposed to find deadlock situations or to prove that there was no way

for deadlock to occur. The types of models employed were graph models, precedence graphs, and petri nets. After it had been determined how to construct deadlock free parallel programs, performance studies of parallelism began to appear.

The models of performance typically were queueing models used to study the effect parallelism, or more specifically multiple processors, had on such performance measures as throughput, utilization, and response time. The effect of the scheduling discipline, the CPU service time distribution, and the types of jobs on the throughput of a multiprocessing system has been studied [5]. It is shown that no significant gain in throughput is obtained by the simultaneous execution of parts of a program on different processors, if the degree of multiprogramming is high enough. However, the response time is improved by multiprocessing. A comparative analysis of several computer organizations such as multiprogrammed systems, multifunction machines, vector machines, array processors, and shared memory multiprocessing systems has been conducted [28]. The conclusions are that multiprogramming increases throughput but response time may deteriorate. Both throughput and response time are improved by all of the other performance enhancement techniques (i.e. shared memory multiprocessing systems, array processors, etc.). However,

to obtain an n-fold increase in throughput, at least an n-fold increase in the effective CPU rate, the effective I/O rate, and the effective bandwidth of main memory is required. These two studies are representative of the types of modeling studies conducted for multiprocessing systems.

Very few performance studies of distributed computer systems (as defined in this paper) exist. A method which uses petri nets and probabilistic grammars has been proposed but not actually carried out [29]. It is claimed that petri nets can be used with probabilistic grammars to produce a composite machine. A composite machine is a Markov process. Standard methods for solving Markov processes can be used; also, techniques associated with the probabilistic grammar can be used to obtain additional performance information. The performance index which can be obtained from such a model is the expected cost of a process given the structure of the process (i.e. its computation and message characteristics).

2.2 MEASUREMENTS OF SYSTEMS

Measurements of computer systems are taken either to be used in conjunction with a model or to gather performance statistics about the system. The usual type of quantities measured for a model are the parameters. The

model then provides the performance information. Models can be used to study trends, tradeoffs, and predict performance behavior as the values of the parameters are varied. Measurements taken of a system, on the other hand, simply indicate how that system performs.

2.2.1 Measurements for Models

Very few modeling studies have been conducted in which actual measurements of a system have been used in the model. It has been shown how to derive parameters typically used in queueing network models from data obtained from the measurement techniques employed by commercially available systems [24].

Most of the measurements supplied by commercial systems are obtained from monitoring techniques, designed to observe the operation of a computer system over a period of time, and record the values of certain variables considered to be significant for evaluating system operation. There are two types of monitoring techniques: event trace and sampling. Both event trace and sampling monitors record system states from specified registers and memory locations. Event trace monitors collect information when a particular event occurs; sampling monitors record data at specified time intervals. The event trace monitor usually obtains more detailed information over a shorter period of time. Meas-

urement monitors can be implemented in many ways.

The implementations of both sampling and event trace monitors can be classified into three types: hardware, software, and hybrid. Hardware monitors are devices attached to various points in the digital circuitry to measure status or count events. Software monitors are measurement routines inserted into the system software to measure the same quantities. Hybrid monitors are a combination of these two types. Measurement data is moved by software routines into special registers which are then read and recorded by hardware monitors.

The monitoring techniques are used by most operating systems to obtain processor and I/O channel utilizations and job statistics over a period of time. It has been shown how to use this data to obtain estimates for queueing network parameters such as CPU Mean Service Time, I/O Device Mean Service Time, and I/O Device Routing Frequencies.

2.2.2 Measurements of the Ethernet

Measurement studies of single processor systems are numerous. However, not many measurements have been taken of distributed systems. These systems are more complicated in that the system state consists of the combined states of all processors in the system. An exam-

ple of the method and measurements obtained for the Ethernet, a local area network, is described below.

The Ethernet is a local network which uses a shared, multi-accesss bus with distributed control as the communications medium [23]. A carrier sense, multiple access with collision detection protocol is employed by the bus to transmit packets of information. When a station (computer) wishes to use the bus it senses the carrier to determine if a transmission is in progress. If no other station is transmitting, the sender can use the bus immediately, otherwise it waits until the packet has passed. It is possible for two stations to begin transmissions simultaneously, producing a collision. Each sender continues to monitor the cable and is able to detect a collision by noticing that the signal on the cable does not match its own output. Each station then waits for a random period of time before retransmitting. A study characterizing the actual behavior of the system was conducted to aid in understanding the system [25].

The broadcast nature of the Ethernet system allowed for the passive collection of measurements. One station can receive all the packets and by using specialized test and monitoring programs can analyze the data collected. The measurements obtained were: (1) number of packets in error, (2) overall traffic characteristics, (3) utilization of the bus, (4) the packet length dis-

tribution (packets are of a variable length), (5) host traffic patterns, (6) interpacket arrival times, (7) the latency and collisions detected by a sender, (8) the overhead of communication in terms of the percentage of control bits versus data bits sent and, (9) the number of packets transmitted to other networks. These measurements were collected under normal operating conditions. Test programs were constructed to generate artificially high levels of traffic in order to observe system behavior when the bus is highly utilized.

2.3 SUMMARY

The studies of parallelism for multiprocessing architectures discussed above indirectly study parallelism. Parallelism, or multiple processors, along with scheduling algorithms, and the amount of memory were all varied to observe the resulting changes in throughput and response time. The approach outlined for studying the performance of distributed computer systems would obtain the cost of a process. The performance study undertaken in this thesis attempts to study parallelism directly; that is, the performance measure to be obtained is the percentage of time processors are actually executing simultaneously.

Methods for obtaining parameters for queueing network models from data collected by monitoring techniques

have been proposed. These monitoring techniques have been implemented on single CPU computer systems. They will have to be modified if they are to be used on a loosely coupled local computer network. A technique employing passive monitoring of a shared bus has been used to take measurements of the Ethernet, a local computer network.

CHAPTER 3

DESCRIPTION OF THE FACILITY

An experimental facility composed of a local computer network referred to previously as the XDCS system, and a host computer, exists for studying the characteristics of distributed computer systems. The hardware and software were designed so that features of the facility could be changed or varied with a minimum amount of effort. The experimental facility can be divided into three parts: (1) the hardware of the XDCS system, (2) the software of the XDCS system, and (3) the operation of the facility. After these three components are described, the features of the XDCS system which are important to subsequent chapters will be discussed in more detail.

3.1 OVERVIEW

3.1.1 Hardware

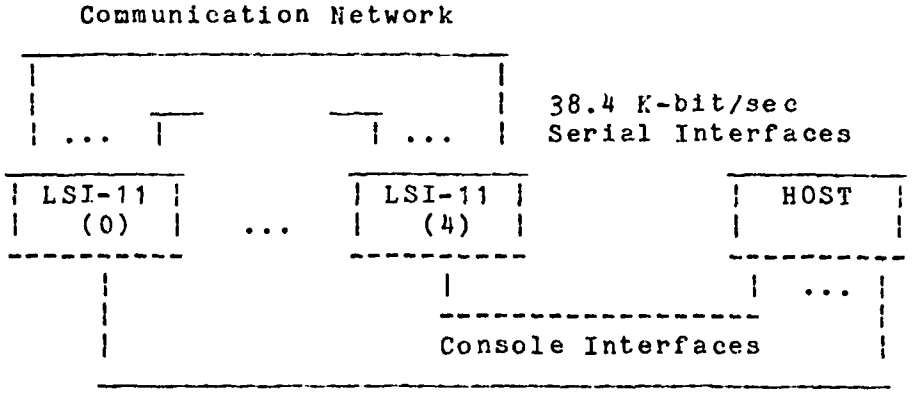
The hardware consists of five LSI-11s linked together by two different communication networks. The first communication network is a Direct, Dedicated and Complete (DDC) interconnection structure [1]. This network was implemented using commercially available components. The second type of interconnection structure is a bus with a CSMA/CD (carrier sense, multiple access

with collision detection) protocol; it is an example of a Direct Shared Bus (DSB) interconnection structure with decentralized control. These two configurations are illustrated in figure 3-1.

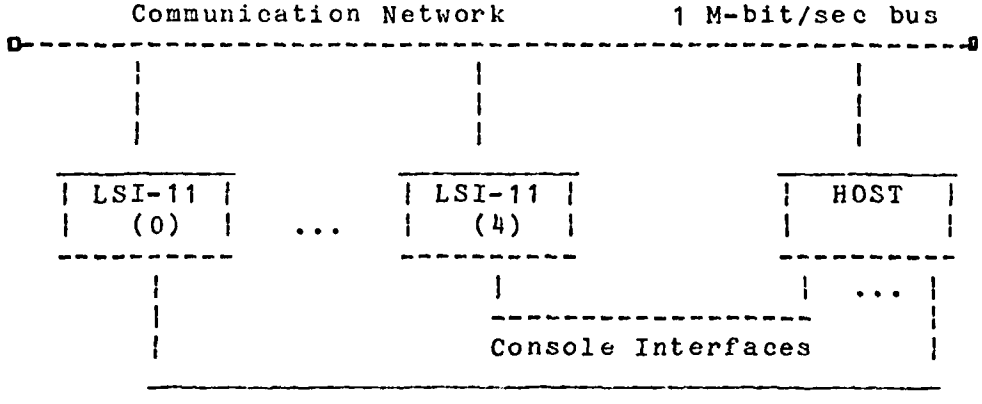
3.1.2 Software

Programs written in EPL, an experimental programming language, are executed on the hardware. EPL is a language which contains the programming language concepts necessary for constructing software to execute in a distributed environment. EPL forces programs to be written as a collection of autonomous processes that may potentially execute in parallel. Since each process only has access to its own variables, processes must communicate with each other and do so through the transmission of unbuffered messages. These messages also supply a means of synchronizing processes.

EPL programs require the runtime support of an operating system kernel. The kernel provides primitives to implement the EPL commands, and decides which computer a process will execute on if the EPL process does not specify this information itself. Thus, the functions of the kernel are to provide process and memory management for EPL programs. The operating system kernel is decentralized in the following way. An identical kernel resides on each processor; however, each kernel knows the



DDC Configuration



DSB Configuration

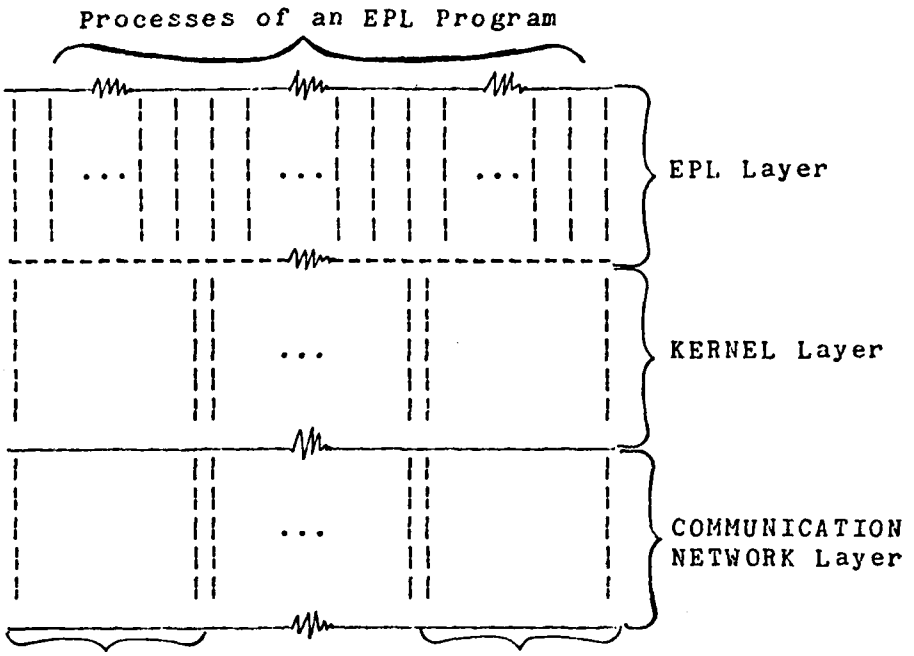
Figure 3-1

state information of only the processes executing on its processor. The creation of processes and the transmission of messages requires state information from other kernels. A communication protocol obtains and updates the necessary state information. A detailed discussion of the design and implementation of this operating system kernel is contained in [13].

The software architecture is organized into three layers as shown in figure 3-2. The top layer is the processes comprising an EPL program. These processes see a single virtual machine provided by the kernel, the second layer; the network of computers is transparent to the individual processes of an EPL program. The communication network layer encapsulates the features of the hardware interconnection structure and is responsible for routing kernel requests for communication to the correct processor. The software for the DDC and DSB configurations is different in only this third layer.

3.1.3 Operation of the Experimental Facility

The local network, XDCS, is hosted by a general purpose computer. Software is constructed on the host system and down-line loaded to the processors of the network. In addition to the standard set of compilers, assemblers, and loaders, facilities exist for compiling EPL programs and interacting with programs executed on



Software Architecture

Figure 3-2

the XDCS system.

The host computer is interfaced to the XDCS network in many ways. The console devices of each LSI-11 microcomputer are connected to serial interfaces of the host system. This allows the XDCS system to be operated from a single terminal attached to the host system. A high speed serial interface is connected from each LSI-11 to the host. This special interface was designed for measurement purposes. The host and the processors of the local network are also attached to the shared bus. A

programmable clock residing on the host system provides centralized timing information that is available to each LSI-11 and/or the host for performance measurements.

3.2 XDCS SYSTEM FEATURES

Messages flow between the entities shown in each of the three layers of software (figure 3-2). In order for a message to be transmitted from one EPL process to another, the respective kernels are called and a communication protocol is invoked. Roughly three kernel level messages must be transmitted to guarantee delivery of an EPL message. The communication network layer is called to transmit each kernel level message. Again, a protocol is followed to guarantee safe transmission of each kernel level message. The protocol implemented in the communication network level will be referred to as the link level protocol for the remainder of this paper. A major part of the modeling process involves representing this link level protocol.

3.2.1 Link Level Protocol

The DDC configuration of the local network is used in this performance study. The interfaces between computers are implemented by serial devices that contain transmitter and receiver status and buffer registers for each interface on each processor. Due to the fact that the interfaces are operated as half-duplex channels, it

is possible for two computers to be attempting communication with each other at the same time. This condition will be defined as contention for later use. Another potentially dangerous situation can occur if processor A attempts communication with processor B, processor B attempts communication with processor C and processor C attempts communication with processor A. This event and other similar situations will be called preemption, due to the fact that one or more processors may have to postpone their communication needs so that anothers and eventually theirs may be satisfied. The link level protocol resolves these potentials for deadlock in addition to initiating and carrying out communication under "normal" circumstances.

In order to resolve the contention and preemption cases, processors are ordered hierarchically by processor number, with processor 0 having the highest priority, and the processor with the largest processor number having the lowest priority. When a contention or preemption situation occurs, the processor with the highest priority is allowed to complete its communication first. Characteristics of the EPL software make it very unlikely that starvation of the lowest priority processor would occur.

Diagrams [27] illustrating the link level communication protocol are shown in figure 3-3. After each

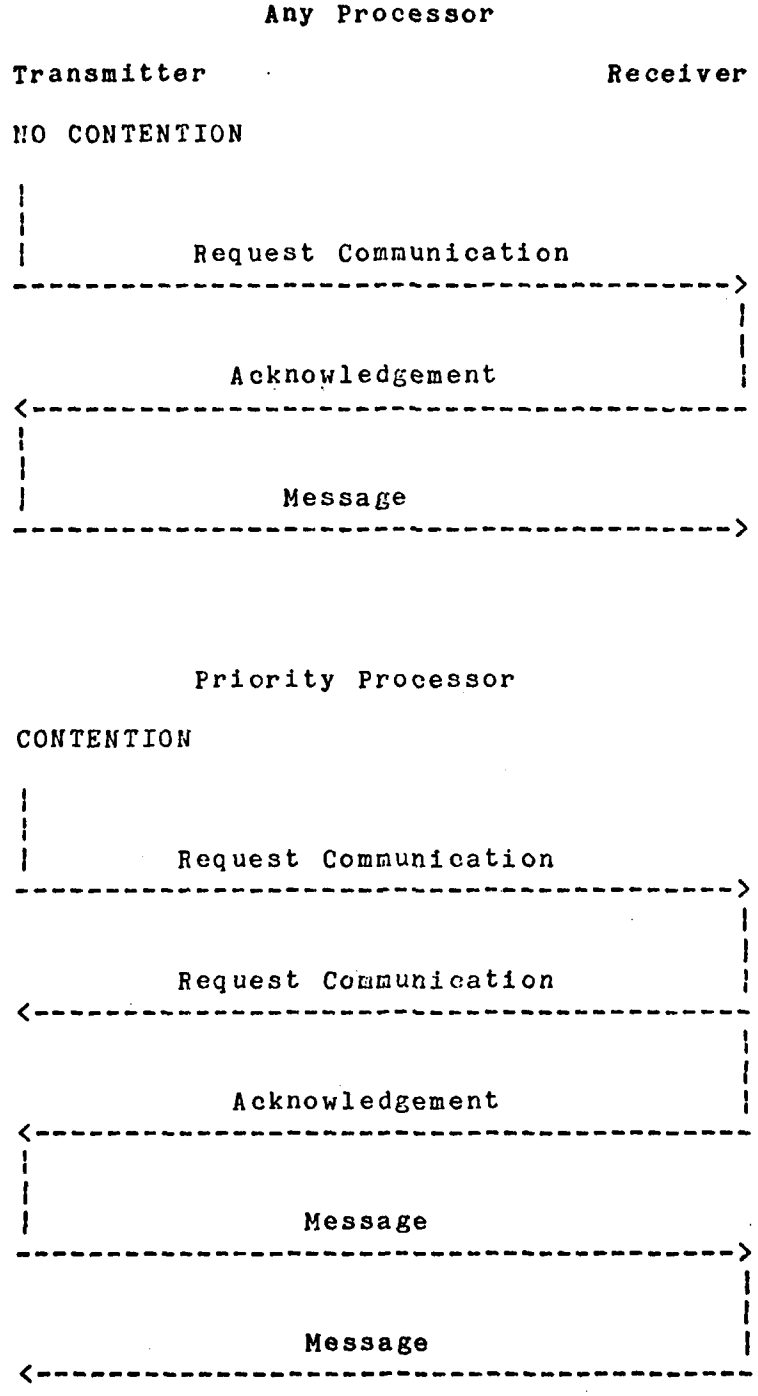

Any Processor
Transmitter
Receiver
NO CONTENTION
Request Communication
Acknowledgement
Message
Priority Processor
CONTENTION
Request Communication
Request Communication
Acknowledgement
Message
Message

Figure 3-3

Non-priority Processor

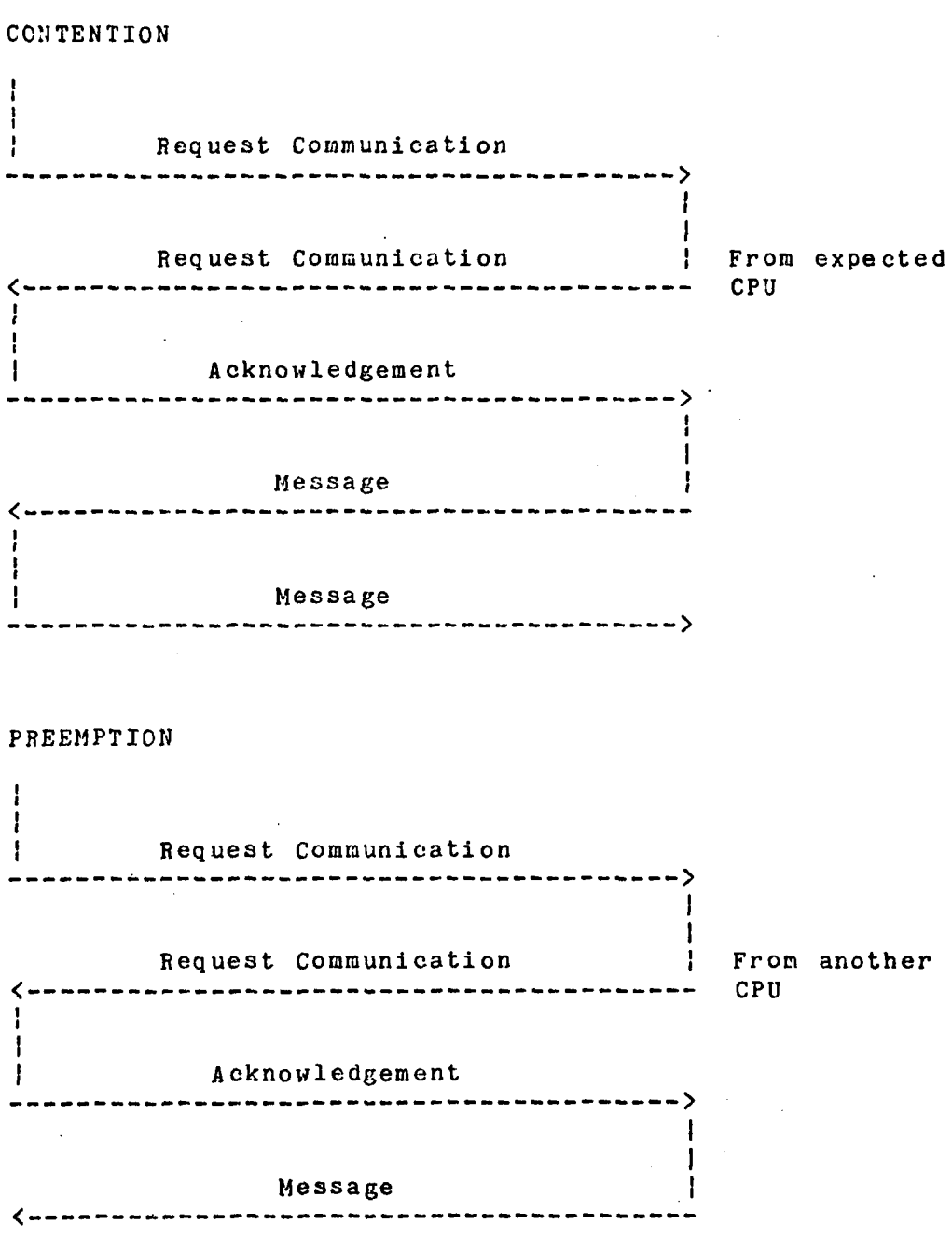


Figure 3-3

kernel call, the communication channels are polled to see if data has arrived from another processor, unless the kernel call resulted in a request to transmit a message. The portion of the protocol labelled "Receiver" corresponds to the processor polling and the action taken if data is detected; the "Transmitter" part of the protocol is used to transmit a message to another processor.

A routine called TRANSMIT, which is part of the communication network software, is called with the destination process name whenever the kernel needs to transmit a message. The TRANSMIT routine determines where the destination process resides. If the source and destination processes are on the same machine, then the communication is called local and there is no need to invoke the "Transmitter" protocol. When the destination process resides on another processor, then the appropriate message information is entered in a queue. A routine, NETWORK, which implements the "Transmitter" protocol is called if the queue contains only the message just entered. If the queue contains more than one message, then the TRANSMIT routine merely returns control to the routine which called it. Once the NETWORK routine is invoked, it empties the queue of all messages.

CHAPTER 4

THE MODEL OF PARALLELISM

Models of systems are usually constructed to predict the values of performance indices and/or to study the tradeoffs or relationships between these indices. Once it has been decided which performance indices are to be studied, the factors or components of the system affecting the specified performance indices need to be identified. Next a representation or model capturing these factors is formulated. All models contain input parameters. Input parameters are variables which incorporate the factors identified as affecting the performance indicies under study. The input parameters can be measured and then used to solve the model to produce one specific solution. It is expected that the output from the model will reflect changes in the values of the parameters. Input parameters can also be compared or related to each other algebraically. This exercise usually provides some insight into system behavior under many different parameter values or operating characteristics. Finally, if possible the model is validated to make sure the representation of the factors is correct.

The model discussed below explains the degree of parallelism exhibited by the XDCS system as it executes

benchmark programs. Although a high potential for parallel execution of the software exists, it is degraded by the need for interprocessor communication. Thus, there is a tradeoff between parallel execution of processors and communication between processors.

4.1 FACTORS INFLUENCING PARALLELISM

4.1.1 Number of Processors

The number of processors used during the execution of a program is an important factor; it is the basic hardware resource providing the potential and maximum bound for parallelism. As a given program is executed on an increasing number of processors, intuitively, it should execute faster; the same number of processes would be spread among more computers. For example, if two processors are used instead of one to execute the same program, the execution time could theoretically be cut in half due to the fact that the number of computers was doubled. This is the maximum amount of decrease in execution time acheivable; it is never reached because of the overhead required to manage the communication between processors.

4.1.2 Bandwidth of Communication Medium

Another factor affecting parallelism is the communication bandwidth. The speed of the communication medi-

um directly affects the amount of time it takes to transmit a message from one processor to another. With serial links, if the baud rate were increased from 1200 baud to 9600 baud, the time to transmit the same data would be shorter. This factor is inversely related to the degree of parallelism. As it takes longer and longer to communicate, the delays due to communication are increased which decreases the amount of time a processor is performing computations.

4.1.3 Frequency of Non-local Messages

The rate at which messages are transmitted to other processors affects how often the communication network software must be invoked. Messages sent between processes located on the same processor do not require the services of the communication network software. Thus, it is seen that the message frequency depends on two components: (1) the number or frequency of EPL messages, and (2) the scheduling algorithm of the kernel. A different EPL program or a different scheduling policy in the kernel would be reflected in the frequency of non-local messages.

4.1.4 Length of Messages

The length of a message, whether it is 5 bytes or 256 bytes long, determines how long it takes to transmit that message from one processor to another. It is pri-

marily the EPL program that determines the message length, although the kernel has some control messages of a constant length; the set of messages transmitted consists of kernel control messages and actual EPL messages which are prefixed by some control information from the kernel.

4.1.5 Message Processing Time

This factor indicates how much time is spent processing a message that has arrived from another processor. The longer it takes to process a message, the less time there is available for the processor to execute the application program.

4.1.6 Time to Implement Link Level Protocol

Another factor which can be identified is the time it takes the protocol to set up for a non-local communication after the kernel requests to transmit a message. This factor has an inverse relationship to the degree of parallelism; the longer it takes to set up for communication, the less time there is available for computation.

4.2 FORMULATION OF THE MODEL

In defining what is meant by parallelism for this study, it was first necessary to define the various states a processor could assume. Three states were seen

to be important; a processor could be: (1) executing EPL or kernel code, (2) waiting for communication with another processor to begin, or (3) communicating with another processor. Two processors are defined to be executing in parallel if they are both executing EPL or kernel code. A global or system state can be attained by combining the local states of each processor in the system. For example, if there are two processors, there are 3^2 or 9 possible system (global) states. Out of these 9 possible states, however, only 3 actually occur. It is impossible for the state, one processor executing EPL or kernel code and the other communicating, to occur. The communication protocol requires the cooperation of 2 processors for communication to take place. Thus, the maximum number of system or global states is 3^n where n is the number of processors in the system.

A Markov model can be used to represent the system, where the states of the model correspond to the system states that occur. The model is a discrete state, discrete time, homogeneous Markov process. The assumptions of this type of Markov process are the following: (1) the transition to the next state depends only on the current state which implies that the time spent in a given state is geometrically distributed, and (2) the transition probabilities are stationary with time [16]. Furthermore, it is assumed that there are an unlimited

number of processes so that a processor is never idle.

4.3 DESCRIPTION OF THE MODEL

The model parameters are listed in Table 4-1. The first parameter, $1/\lambda$, is the average interarrival time between non-local messages, which corresponds to the factor, frequency of non-local messages. $1/\mu_t$, is the average time to initiate communication with another processor. This parameter corresponds to the factor previously referred to as the time to implement the link level protocol. $1/\mu_o$, the average time to send a message, and $1/\mu_i$, the average time to receive a message, both reflect the factors, bandwidth of the communication medium and message length. $1/\mu_p$, the average time to process a message is the factor, message processing time. The last four parameters, p, pr, rnc, and rc are used to further define the transition probabilities between states and will be described in more detail below. A different Markov model exists for different numbers of processors due to the state definition. For example, the maximum number of global states for two processors is 9, while for three processors it is 27. Thus, the factor, number of processors, is reflected by different values in the parameters, $1/\lambda$, p, pr, rnc, and rc for each model, and the actual system states possible in each model. Figure 4-2 illustrates the two processor model and Figure 4-3, the three processor model.

Table 4-1

Model Parameters

$1/\lambda$	-	average time between non-local messages
$1/\mu_s$	-	average time for protocol setup
$1/\mu_o$	-	average time to output a message
$1/\mu_i$	-	average time to input a message
$1/\mu_p$	-	average time to process a message
p	-	probability of contention
pr	-	probability of preemption
rnc	-	probability there is another message to transmit after sending a "normal" message
rc	-	probability there is another message to transmit after sending a contention message

4.3.1 Two Processor Model

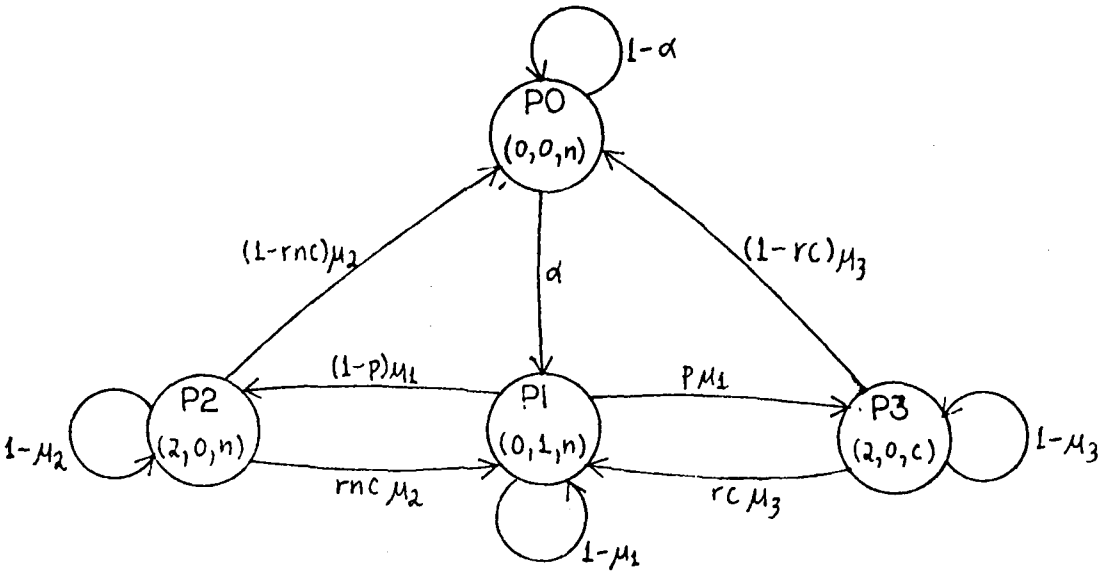
The states of the model are represented by a 3-tuple, (NC,NW,s). The first element, NC, indicates the number of processors communicating; NW is the number of processors waiting to communicate, and s is a member of the set S = {c,p,n}. The members of the set, S, correspond to contention, preemption, and normal communication modes. It is necessary to distinguish between these three modes because the time to send, receive, and process message(s) in each case is quite different.

For two processors, a maximum of 9 global states is possible, but only 3 states actually occur. The model has 4 states, however. This is due to the fact that contention can occur in the state where both processors are communicating. Thus, two states are required to represent this condition; P2 is the state where both processors are communicating under normal conditions, and P3 is the state where both processors are communicating in contention mode.

As stated earlier, the state time distributions are assumed to be geometric, which is of the form (1-q)q. This means that the probability of leaving a state is q, and the average time spent in a state is 1/q. It is possible to combine the parameters to approximate the average state time, 1/q, which can then be inverted to define q, the transition probability out of a state. The parameters p, rc, and rnc are used to determine the specific transition probabilities when transitions from one state to two or more states are possible (e.g. from P1 either state P2 or state P3 can be entered).

4.3.2 Three Processor Model

The three processor model is similar to the two processor model in that the parameters, and the 3-tuple, (NC,NW,s), which distinguishes the states, are the same. The difference between the two models lies in the number



$$\alpha = \frac{1}{\left(\frac{1}{\lambda} - \frac{1}{\mu_1} - \frac{1-p}{\mu_0} - \frac{p}{\mu_3}\right)} \qquad \mu_2 = \frac{1}{\left(\frac{1}{\mu_i} + \frac{1}{\mu_p}\right)}$$

$$\mu_3 = \frac{1}{\left(\frac{2}{\mu_i} + \frac{1}{\mu_p}\right)}$$

P0 : both computing (EPL or Kernel)
P1 : one computing, one waiting to communicate
P2 : both communicating, normal mode
P3 : both communicating, contention mode

Figure 4-2

of states possible; the two processor model has only four states while the three processor model has eight states. Also, it is possible for the preemption mode to occur which cannot happen with two processors.

4.3.3 Four Processor Model

It was determined that a four processor model would have 19 states, and a five processor model would have 29 states. Due to the large number of states, and the results obtained from the two and three processor models, which will be discussed later on, it did not seem that much information could be gained from implementing the four and five processor models.

4.4 VALIDATION OF THE MODEL

4.4.1 Procedure

Validation of the model is performed by comparing the state probabilities predicted by the model which are obtained by solving the model, to state probabilities computed from system observation. In order for the model to be solved, values for the input parameters are needed. These values can be obtained by measurement techniques. The state probabilities computed from observation are also obtained from measurement techniques.

Once the system has been instrumented to procure the appropriate measurements, the validation process,

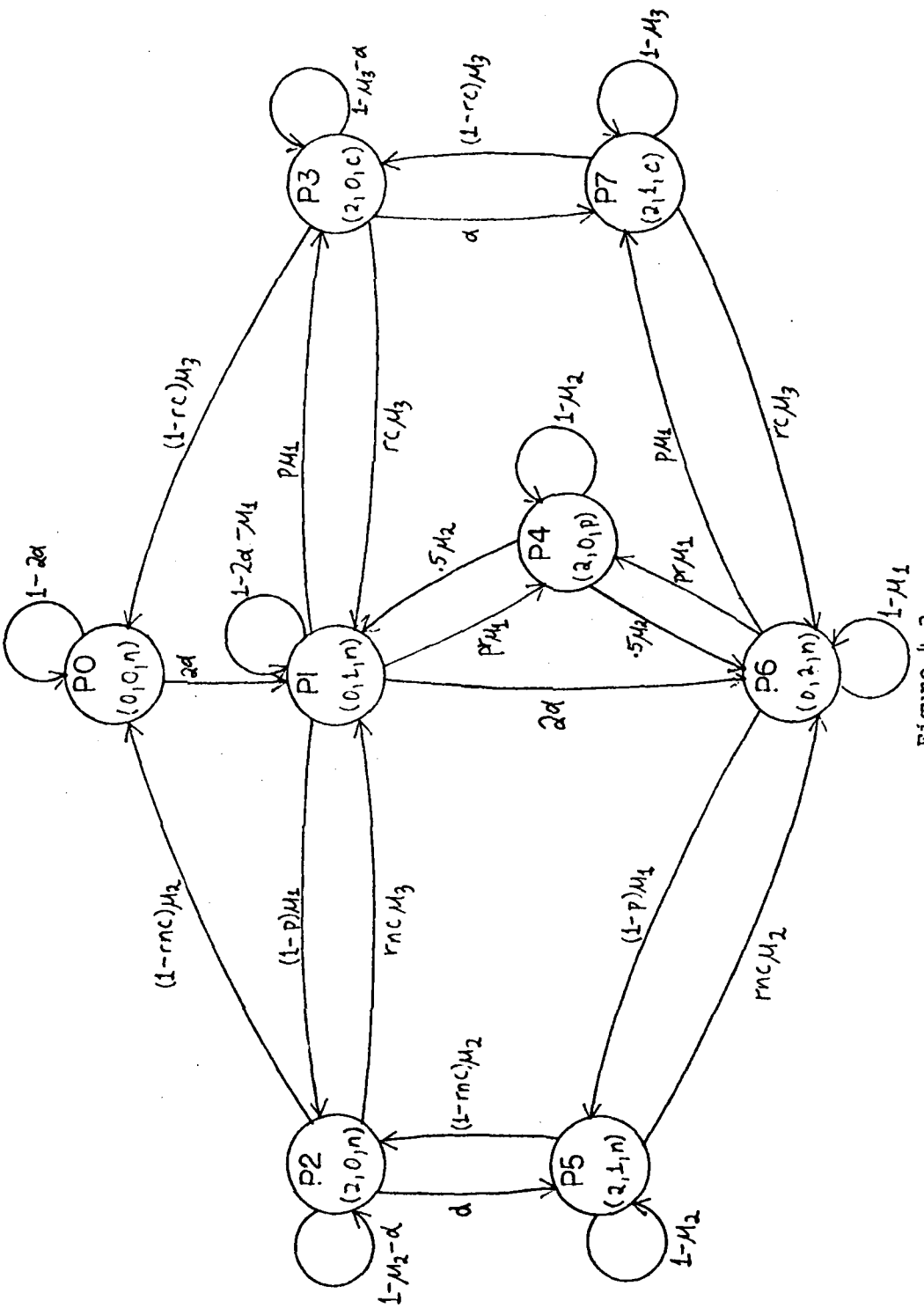

P0
(0,0,n)
P1
(0,1,n)
P2
(2,0,n)
P3
(2,0,c)
P4
(2,0,p)
P5
(2,1,n)
P6
(0,2,n)
P7
(2,1,c)
1-2α
2α
1-2α-μ1
pμ1
rcμ3
(1-rc)μ3
1-μ3-α
α
1-μ3
(1-rc)μ3
1-μ2
.5μ2
pμ1
rμ1
.5μ2
pμ1
rcμ3
1-μ1
(1-rnc)μ2
(1-p)μ1
rnc μ3
(1-rnc)μ2
α
(1-p)μ1
rnc μ2
1-μ2-α
1-μ2

Figure 4-3

$$\alpha = \frac{1}{\left(\frac{1}{\lambda} - \frac{1}{\mu_1} - \frac{1-p}{\mu_0} - \frac{p}{\mu_3}\right)} \qquad \mu_3 = \frac{1}{\left(\frac{2}{\mu_i} + \frac{1}{\mu_p}\right)}$$

$$\mu_2 = \frac{1}{\left(\frac{1}{\mu_i} + \frac{1}{\mu_p}\right)}$$

P0 : all computing (EPL of Kernel)

P1 : 2 computing, 1 waiting to communicate

P2 : 1 computing, 2 communicating, normal mode

P3 : 1 computing, 2 communicating, contention mode

P4 : 1 computing, 2 communicating, preemption mode

P5 : 2 communicating, 1 waiting to communicate, normal mode

P6 : 1 computing, 2 waiting to communicate

P7 : 2 communicating, 1 waiting to communicate, contention mode

Figure 4-3

sometimes referred to as tuning the model, can begin. The validation process consists of obtaining the two sets of state probabilities, comparing them, noting the inaccuracies, proposing a reason for the inaccuracies and subsequently modifying the model or measurements. The process is repeated until the two sets of probabilities are sufficiently close enough so that the model can be used with confidence. Sufficiently close is a term which usually is constrained by the requirements of the performance study.

4.4.2 Types of Inaccuracies

During the validation process many inaccuracies are encountered; they can be classified as belonging to one of two types: (1) model inaccuracies, or (2) measurement inaccuracies. Model inaccuracies can be further subdivided into formulation inaccuracies, and solution inaccuracies. The types of inaccuracies are presented here, but a discussion of the validation process will be delayed until the instrumentation of the system has been described.

Formulation inaccuracies are generally attributable to an incorrect amount of detail in the model, or an incorrect representation of the parameters. There can be either too much detail represented or, going to the other extreme, not enough detail was incorporated. In any

case, the model is not solvable.

Solution inaccuracies may be caused by mistakes in mathematical manipulation or errors occuring in the numerical methods used. Numerical methods are algorithms usually of an iterative nature used to approximate solutions to mathematical problems.

Measurement inaccuracies occur if the measurements are performed incorrectly. In this study, that could happen in two places. The input parameters could be wrong, or the process of gathering and reducing data to obtain the observed frequency of system states could be erroneous.

4.4.3 Benchmark Programs

Two benchmark programs written in EPL were used to validate the model. The first program, CALIBRATE, is a small program with known variables. The program creates a known number of pairs of processes. Each pair of processes transmit and receive a known number of messages. The computation time between messages is drawn from a uniform distribution with a known mean value. This program was especially helpful in tuning the model. The results of validation are presented in Chapter 6, after the instrumentation of the system is discussed in Chapter 5.

The other program, BSORT, is a binary sort routine representative of the type of data and control structures used to decentralize software. The program sorts a known number of random numbers by constructing a binary tree. Each number is sorted as it is placed in the tree. Each node of the tree is implemented by a single process. When all of the numbers have been placed in the tree, it is unloaded using an ordered tree traversal. Loading and unloading the tree requires each node to communicate with one or more of its subtrees. All processes representing one level of the tree are assigned to the next available processing element in a circular fashion.

CHAPTER 5

INSTRUMENTATION OF THE SYSTEM

Instrumenting the XDCS system was done primarily to provide the means for validating the model of parallelism. During the design and implementation of measurements three things must be considered: (1) system resources and characteristics which impact the design, (2) guaranteeing that the measurements are correct, and (3) making sure that the instrumentation does not alter the execution characteristics of the system, or at least accounting for and removing the effects of interference from the measurements.

The measurements needed to validate the model fall into two categories. Values for the parameters of the model are required in order to obtain a solution, and measurements used to calculate the percentage of time spent in each of the possible system states are needed for comparison with the solution produced from the model.

5.1 SYSTEM FEATURES IMPACTING THE DESIGN

5.1.1 Resources

An important resource available for measurements is a programmable real-time clock. It is a device manufac-

tured by Digital Equipment Corporation that can be used with their processors [9]. The clock frequency can be selected along with one of four different modes of operation. Two registers are available for controlling the clock, a control and status register and a data buffer register. In the first mode of operation, an overflow flag is raised and if enabled, an interrupt is generated after a time interval specified by the user has elapsed. The second mode of operation is the same as the first except that after the overflow flag is raised, the registers are reinitialized and everything is repeated until the clock is turned off. A third use of the clock allows the user to start the clock and then read it upon the detection of a specified event. The fourth mode is the same as the third except that after the clock is read it continues to increment and can be read again. This continues until the clock is disabled.

High speed serial interfaces from each LSI-11 to the host computer were installed after the instrumentation of the system had begun. Problems encountered in satisfying the correctness and interference criteria during the instrumentation resulted in the addition of these interfaces. The problems will be described later.

Another resource available for taking measurements is the Break Point Trap. Setting a bit in the processor status word causes a trap to a location in memory after

the execution of every instruction. Instruction counts can be obtained easily. More complex measurements can be taken by turning the trap on and off at selected times.

5.1.2 System Characteristics

The host system, a DEC PDP 11/23, runs the UNIX operating system. Standard compilers, assemblers and loaders along with the EPL compiler are used to construct an executable image of an EPL program and the kernel, which provides the necessary runtime support. Special software has been written which allows the image to be loaded in each of the LSI-11s. All of the kernel code is written in the programming language C except for a small assembler interface. The interface contains code for trap and interrupt vectors and context switching between EPL and the kernel.

Also existing on the host system is a Modula compiler [8]. Along with the compiler there is an assembler program which provides runtime support for Modula programs. This makes it possible to write Modula programs capable of executing in a stand-alone environment.

5.2 MEASUREMENTS OF PARAMETERS

A different model is used for different numbers of processors. The parameters for each of the models, how-

ever are the same; the differences occur in the values obtained for the parameters of different models. A description of the methods used to measure the parameters, followed by a summary of the problems encountered is given below.

5.2.1 Methods

5.2.1.1 Average Time Between Non-local Messages ($1/\lambda$)

$1/\lambda$ is the first parameter that was measured. Three different ways, described below, of measuring this parameter were considered. The parameters were measured on only one processor of the network, processor 0, due to the fact that the clock resided on that processor.

The first method is to use the Break Point Trap to accumulate the number of machine instructions executed during the measurement session. A counter containing the number of non-local messages is also needed. Letting M represent the number of machine instructions executed, A represent the average instruction execution time, and N represent the number of non-local messages, $1/\lambda$ can be calculated using the following equation.

$$1/\lambda = (MA) / N \qquad (1)$$

The second method involves using the programmable real-time clock. The clock is used to generate periodic

interrupts. The interrupts are accumulated as is the number of non-local mersages, N. If I is the number of interrupts, and B is the time between interrupts, then the following equation can be used.

$$1/\lambda = (IB) / N \qquad (2)$$

A third way to measure $1/\lambda$ is to use the third operating mode of the clock. At system startup or the first time a non-local message is transmitted, the clock is turned on. Each time a non-local message is sent, the clock is read. The value of the clock is accumlated in addition to the number of non-local messages. Now, if S represents the clock sum and N has its usual meaning, $1/\lambda$ has the following calculation.

$$1/\lambda = S / N \qquad (3)$$

It was decided to use the third method for a few reasons. The most important reason is that the interference with normal system execution is much less than the first two methods. An interrupt routine is invoked after every instruction when the Break Point Trap is used. The second method exhibits the same problem although there is some control over the amount of inteference. Note that the time between interrupts should be short to obtain more resolution and a more accurate measurement, but the shorter the time between interrupts

the greater the interference becomes. Another reason for choosing the third method is that the calculation and implementation seemed to be more straightforward.

Once the method was chosen, all that remained was to implement it. Although changed later, while the implementation of this measurement was being carried out, the programmable real-time clock resided on processor 0 of the XDCS network. The clock was turned on in the interface at system startup. The NETWORK module was modified to call an assembler routine, MEAS, in the interface which manipulated the registers of the clock and incremented the necessary counters. After experimenting with different clock frequencies, the rate of one tick every millisecond was chosen.

5.2.1.2 Average Time For Protocol Setup ($1/\mu_1$)

This parameter corresponds to the time it takes to initiate communication with another processor. Once the choice of measurement technique was made for $1/\lambda$, it was decided to measure the remaining parameters that are average event times the same way. The difference with this parameter is that the clock needs to be turned on when the remote processor is notified and turned off when the acknowledgement is received. This required the addition of an assembler routine, MINIT, to turn the clock on. Both MINIT and MEAS are called from the ap-

propriate places in NETWORK. The actual time in seconds per setup is the result. The time to setup for communication was a degree of magnitude smaller than the computed rate for $1/\lambda$. Thus, the clock rate was set at 10,000 ticks per second.

5.2.1.3 Average Time to Output a Message ($1/\mu_o$)

The measurement of this parameter is the same as $1/\mu_t$; the MINIT and MEAS routines in the interface are used. The only difference is in the placement of the calls to MINIT and MEAS from NETWORK. The clock rate chosen is one tick every millisecond.

5.2.1.4 Average Time to Input a Message ($1/\mu_i$)

The measurement of this parameter is also the same as $1/\mu_t$. Again the only difference is in the placement of the calls to MINIT and MEAS from NETWORK.

5.2.1.5 Average Time to Process a Message ($1/\mu_p$)

Again, measurement of this parameter is the same as $1/\mu_t$, with the only difference occurring in the placement of the calls to MINIT and MEAS.

5.2.1.6 Probability of Contention (p)

Two counters are needed to compute the probability of contention. One counter contains the number of messages transmitted and the other counter keeps track of

the number of times collisions occur. Dividing the number of collisions by the total number of messages gives the probability of contention, p.

5.2.1.7 Probability of Preemption (pr)

One more counter which keeps track of the number of times preemption occurs is required to calculate the probability of preemption. Dividing the number of preemptions by the total number of messages gives the probability of preemption, pr.

5.2.1.8 Probability of Another Message to Transmit (rnc)

This parameter represents the probability that another message exists in the transmitter queue after a message has been sent under normal circumstances. The calculation of this parameter requires two counters. The first one counts the number of times NETWORK is entered and there isn't any contention. The second one counts the number of messages transmitted after a normal message has been sent. The probability that there is another message to transmit is computed by dividing the second counter by the first.

5.2.1.9 Probability of Another Message, Contention (rc)

This parameter represents the probability that another message exists in the transmitter queue after a contention mode message has been sent. The calculation

of this parameter is the same as the previous one except that the counters required count the number of times NETWORK is entered and there is contention, and the number of messages transmitted after a contention message has been sent.

5.2.2 Problems Encountered

One problem encountered was the inability to obtain the variance for the parameters. Due to the fact that the sum of squares must be maintained, the assembler routines implementing the measurements became so long that there wasn't enough time to take the measurements and receive and process the data sent from other computers. The data loss caused the program to hang indefinitely.

The other problem encountered was the consequence of problems found in implementing the measurements for obtaining the observed state probabilities. In an attempt to correct the problems with the observed state probabilities, the clock was moved from processor 0 of the XDCS network to the host system. This meant that the implementation of the measurements of the parameters had to be changed. It required changes to the kernel, the EPL program, and the communication system. Also, new software had to be written on the host computer to start, stop, and record the measurements.

5.3 OBSERVED STATE PROBABILITIES

The system states are composed of the states of the individual processors. Thus, it is first necessary to determine the local processor state for each processor in the system, and then to correlate that data to finally arrive at the system state. The implementation of these tasks and the problems encountered during the implementation are discussed below.

5.3.1 Processor (Local) States

Sampling techniques are a widely used method of estimating the utilization of various processor states [12]. The primary advantage of sampling techniques is that the amount of data which needs to be collected to estimate quantities of interest is relatively small. Periodically, or randomly, the state of the processor is recorded. After the measurement session, this data can be used to compute the percentage of time spent in each of the defined processor states.

Periodic sampling was chosen to acquire the local states. It was implemented by using the programmable real-time clock to generate periodic interrupts. A variable, SAMPLE, was changed in the kernel everytime a state transition occurred. The interrupt routine transmitted the value of SAMPLE to the host computer. The host computer received the data and stored it in

separate files, one for each computer executing in the local network. A necessary requirement for computing the global state is that the samples taken from each processor must be taken at the same time. This was achieved by connecting the overflow signal from the clock to the external event line for each processor. When the overflow occurs an interrupt is generated on each processor.

5.3.2 System (Global) States

The global states are computed by feeding the files of raw data into a C program. One character at a time is read from each of the files and then it is determined which global state is represented. A counter representing each state is incremented whenever the corresponding state is found. The percentages are calculated by dividing the counts for each state by the total number of samples. A different program is required for different numbers of processors.

Seven different local (processosr) states were defined when the sampling was implemented. This was done to be consistent with other measurements taken of the system; these states had already been defined along with the occurrances of state transitions in the kernel code. It is also felt that the data collected should be as general as possible. Later it can be analyzed in many

different ways through the use of different data reduction programs. The sampling data was more detailed than necessary for this analysis. It presented no problem because states could be combined to arrive at the state definitions required by the model.

5.3.3 Problems Encountered

Numerous problems were encountered with the data acquired from sampling in trying to satisty the correctness and interference criteria. The CALIBRATE program used to validate the model was used here also, with modified characteristics, to verify that the data obtained was correct and to determine the interference caused by sampling.

The first problem encountered was that utilization of the states calculated from data collected from separate experiments run at the same sampling rate exceeded the statistical variation allowed. The source of this problem was the startup procedure for programs executed on the XDCS network. Each computer was started manually by transmitting to the LSI-11 the address of the beginning of executable code. Thus, it was impossible to start the same program the same way twice. This problem was solved by putting a jump to the beginning of executable code in location 0. A global initialization switch on the network transfers control to location 0 on

every computer at the same time when pressed. Now, the same program could be started the same way every time.

The next problem was that data collected at different sampling rates produced results exceeding statistical variation in those states with ten percent or less utilization. It was found that the duration of time in some of these states was as small as one millisecond. Apparently, sampling was not performed fast enough to obtain enough samples in the short states; the state transitions were quicker than the time between samples. The sampling rates experimented with were 77, 91, 100, 143, and 333 samples per second. It was decided to sample at rates in the vicinity of 7000 to 11,000 times per second.

Sampling at very high frequencies generated more problems. First of all the host system was unable to collect the data due to the overhead of the operating system. Secondly, sampling changed the behavior of the software in two ways. The time spent in the states involving only computation was increased by an amount directly related to the sampling period. The time spent in the states of the processor involving use of the communication network was modified in a more complex way. Sampling does not disturb I/O under control of hardware separate from the CPU. The cost of sampling is 38 microseconds. The measurements of the states involving

only computation were adjusted using this value and the number of samples. The calibration program shows that with this adjustment, sampling correctly measures the short states involving only computation. It is not clear, however, how to adjust the measurements of the states involving I/O.

The solution to these problems involved hardware and software modifications. The programmable clock was moved from processor 0 of the XDCS system to the host system. High speed serial interfaces from each LSI-11 to the host machine were installed. Every time a state transition occurred in the software executing on the local network, a character code was transmitted to the host system over the high speed interfaces. Sampling could be controlled from the host machine by generating periodic interrupts using the clock. The interrupt routine would then read all the I/O ports from the LSI-11s and either store or reduce the data. It was decided to write software to obtain the measurements in Modula in order to be able to respond as quickly as possible. This implementation of sampling does not interfere with the execution of the XDCS system.

CHAPTER 6

RESULTS

After the model had been formulated and the system instrumented for measurements, the validation process could begin. A description of that process and the results obtained are presented below. Information obtained from the model is then discussed.

6.1 VALIDATION PROCESS

During the validation process, the errors encountered were primarily with the model. Although the states of the model were correct, the parameters and hence the transition probabilities were erroneous. The measurements of the parameters, however, were performed correctly. Some discrepencies due to the careless definition of the states used for sampling were discovered.

The biggest mistake was in not correlating the transition probabilities with the average time spent in the states of the model. For example, initially, it was assumed that the transition from P0 to P1 (see Figure 4-2) was λ. This was not correct. The average time spent in the state P0 was the time the system was not involved in servicing messages or initiating communication with another processor. Thus, the average time in

state P0 was corrected to be

$$\alpha = 1/\lambda - 1/\mu_1 - (1-p)/\mu_0 - p/\mu_3.$$

The state P1 was also erroneously represented at first. It was assumed that a transition to P2 or P3 would occur when a processor polled its communication channels and found data from another processor. This was grossly incorrect. The time spent in state P1 was actually the time needed to send a byte initiating communication and wait for an acknowledgement.

Another mistake was that not enough detail was represented in the model. Initially, the parameter $1/\mu$ was used to represent the time to send one message, and thus, it was assumed that $2/\mu$ was the time to send two messages (in contention mode). It was not that simple, however. The protocol was slightly different in the contention mode and was not merely twice the time incurred in the normal mode; also, the processing time of the message needed to be included. Therefore $1/\mu$ was broken into three parameters, $1/\mu_i$, the average time to input a message, $1/\mu_o$, the average time to output a message, and $1/\mu_p$, the average time to process a message. Now the average service time for one message is $(1/\mu_i + 1/\mu_p)$ and the average service time for two messages is $(2/\mu_i + 1/\mu_p)$. $1/\mu_o$ was required for the calculation of α, the transition from P0 to P1. The other place where

there was not enough detail, was in calculating the probability of there being another message in the transmitter queue after a message had been sent. The parameters rnc and rc were originally represented by one parameter, r. It was then discovered that the probability of another message in the queue was quite different in the contention and non-contention cases.

Mistakes in gathering the sampling data were found. These mistakes were primarily due to carelessness and oversights in identifying the occurances of state transitions in the kernel and communication network code. The state transitions had to be precisely defined so that they corresponded to the states of the model.

The errors outlined above were the major ones causing large discrepencies between the model and the observed state probabilities. Many changes to the parameters and to the combinations of the parameters were made before the final representation was reached.

6.2 VALIDATION RESULTS

Two programs were used to validate the two processor model, CALIBRATE, and BSORT, a binary sort program. Only the BSORT program was used to validate the three processor model. Table 6-1 contains the results from the CALIBRATE program, Table 6-2 the results from BSORT for two processors, and Table 6-3 the results from BSORT

for three processors. The relative error between the model and the measured data in the two processor case is 20% or less. In the three processor case, however, the relative error is 20% to 30% except for one state which is 55%. Composite states, however, exhibit much smaller relative errors. This is illustrated in Figure 6-4 which shows the percentage of time two, one, and zero processors are computing (vs. performing I/O) when executing the binary sort algorithm, BSORT. The graph shows both the models predictions and the measured values, first for two processors and then for three. The differences between the model and the measured values shows that the model is on the conservative side; the measured values show a higher degree of parallelism.

A reason for the model's inaccurracy is that this model captures the communication patterns between processors, but not the communication patterns between processes. This point can be shown by examining the three processor model (Figure 4-3). When this model was first formulated, the transitions from P5 to P2 and P7 to P3 were not present. It was assumed that P5 would transition to P1 as would P7. However, due to the structure and scheduling of the processes it is possible, and actually was the case that states P2 and P3 were entered. Most likely, this is due to the fact that the waiting processor was trying to initiate communica-

Table 6-1

VALIDATION RESULTS

CALIBRATE

VALUES OF PARAMETERS		α	0.001330
λ	0.001278	μ_2	0.013267
μ_1	0.088192	μ_3	0.010052
μ_0	0.053999	rnc	0.000000
μ_i	0.041480	rc	0.000000
μ_p	0.019506	p	0.006579

STATE PROBABILITIES			
State	Model	Observed	Relative Error
P0	0.896450	0.905217	0.009684
P1	0.013516	0.014783	0.085707
P2	0.089254	0.079130	0.127941
P3	0.000780	0.000870	0.103448

Table 6-2

VALIDATION RESULTS

BSORT

VALUES OF PARAMETERS		α	0.012206
λ	0.010620	μ_2	0.013752
μ_1	0.090966	μ_3	0.010137
μ_0	0.055364	rnc	0.000000
μ_i	0.038563	rc	0.554770
μ_p	0.021374	p	0.693856

STATE PROBABILITIES			
State	Model	Observed	Relative Error
P0	0.078933	0.101190	0.219960
P1	0.099558	0.095982	0.037252
P2	0.201614	0.231399	0.128716
P3	0.619896	0.571429	0.084817

Table 6-3

VALIDATION RESULTS

BSORT (3 Processors)

VALUES OF PARAMETERS		α	0.013885
λ	0.008268	μ_2	0.013397
μ_1	0.070508	μ_3	0.009859
μ_0	0.055082	rnc	0.000000
μ_i	0.037335	rc	0.470588
μ_p	0.020895	p	0.199225

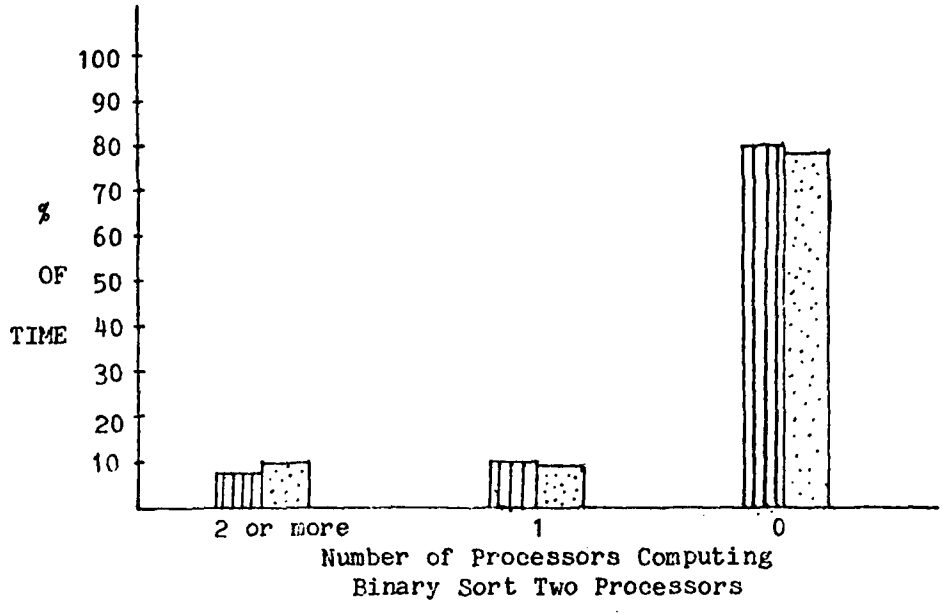
STATE PROBABILITIES			
State	Model	Observed	Relative Error
P0	0.140149	0.148012	0.053121
P1	0.041782	0.057437	0.272570
P2	0.272515	0.354934	0.232208
P3	0.046180	0.036819	0.254252
P4	0.000000	0.000000	0.000000
P5	0.378870	0.243741	0.554397
P6	0.022880	0.030993	0.260213
P7	0.097623	0.128130	0.238089

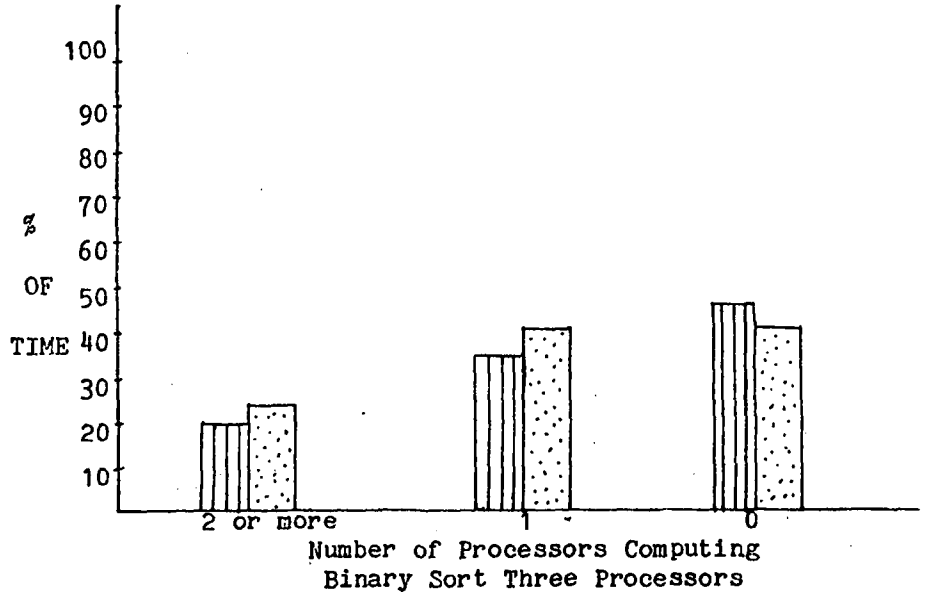
tion with the processor that was receiving the message. If, on the other hand, the waiting processor was trying to communicate with the sender, state P1 would be entered. This situation shows the type of information that cannot be represented in this model. All that is known is the number of processors involved in some event, but there are many ways the event can occur. The different ways an event can occur depends upon the structure (i.e. the computation and communication characteristics and patterns) of processes.

6.3 MODEL PREDICTIONS

One point brought out by the model is that parallelism is increased by the addition of a processor. Figure 6-4 shows the percentage of time two, one, and zero processors are computing (vs. performing I/O) when executing the binary sort algorithm, BSORT. There is more parallel activity when three processors are used rather than two.

Figure 6-5 shows the percentage of time two or more processors would execute simultaneously for four different values of λ while the other parameters were held constant. The percentage of time two or more processors are excuting in parallel is state P0 in the two processor model and states P0 and P1 in the three processor model. The percentages are approximately the same for


100
90
80
70
60
50
40
30
20
10
%
OF
TIME
2 or more
1
0
Number of Processors Computing
Binary Sort Two Processors


100
90
80
70
60
50
40
30
20
10
%
OF
TIME
2 or more
1
0
Number of Processors Computing
Binary Sort Three Processors


- Model Results
- Measured Results

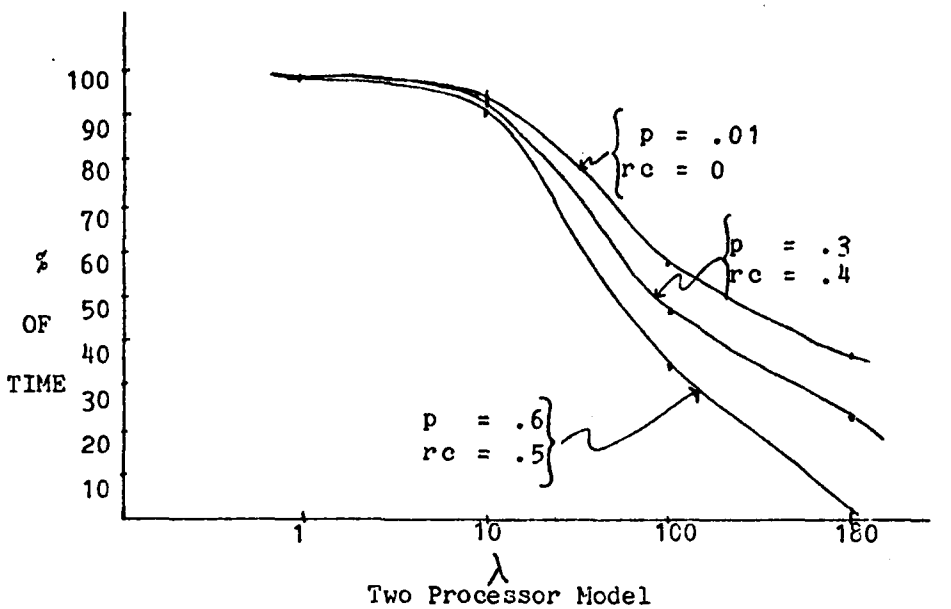
Figure 6-4

the same values of λ regardless of the number of processors used. The values are more robust in the three processosr case due to the fact that two states represent two or more processors executing in parallel.

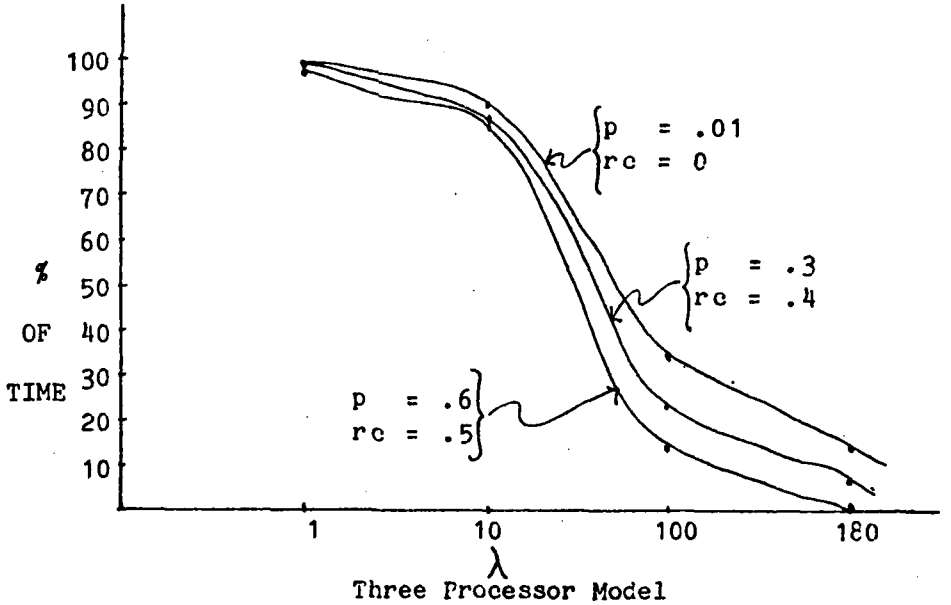
Figure 6-6 shows the percentage of time two or more processors would execute simultaneously for four different values of μ_1 while the other parameters were held constant. The degree of parallelism is not as high when three processors are used instead of two for the same values of μ_1.

Figure 6-7 shows the percentage of time two or more processors would execute simultaneously for four different values of the three parameters, μ_o, μ_i, and μ_p. These three parameters were changed at the same time because they all depend upon the message length and the bandwidth of the communication medium. Thus, a change in either of these two factors would cause a change in all three parameters. Again, the degree of parallelism is lower for three processors for the same values of μ_i, μ_o, μ_p.

Adding another processor will not improve the degree of parallelism if the values of the parameters remain the same, in fact it may degrade it due to the fact there is a higher probability of one or more of the processors wanting to communicate. However, it has been

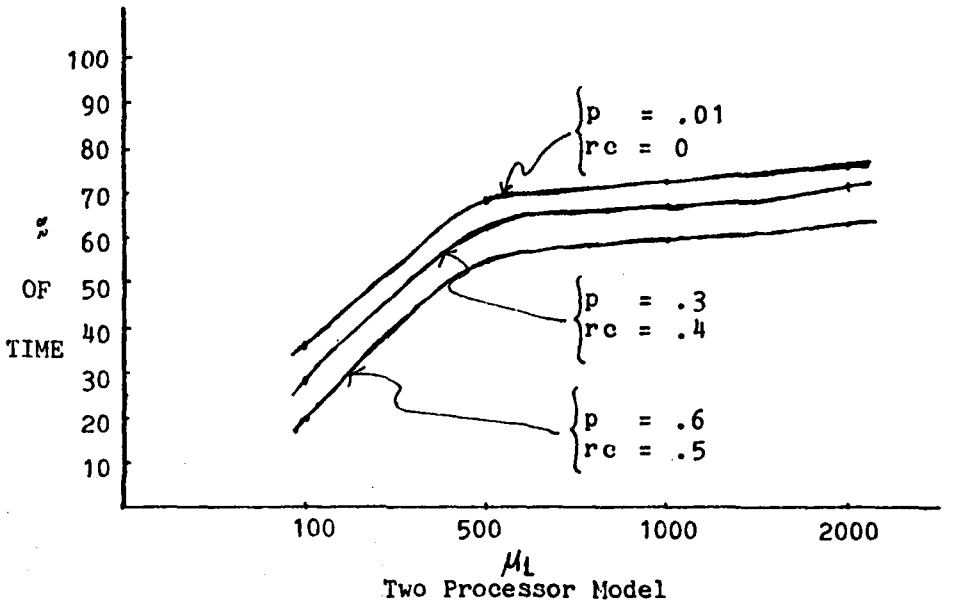


Two Processor Model

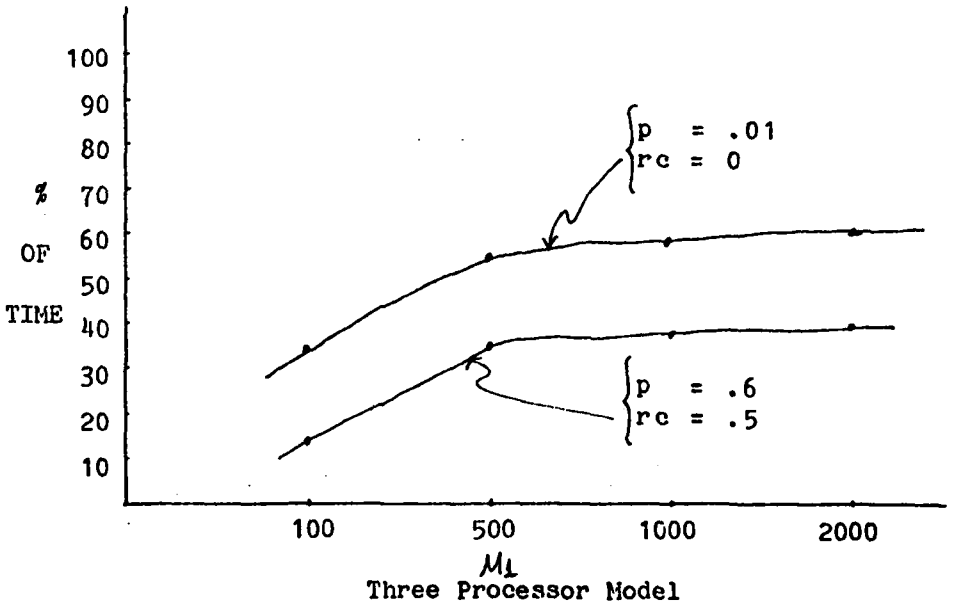


Three Processor Model

Figure 6-5

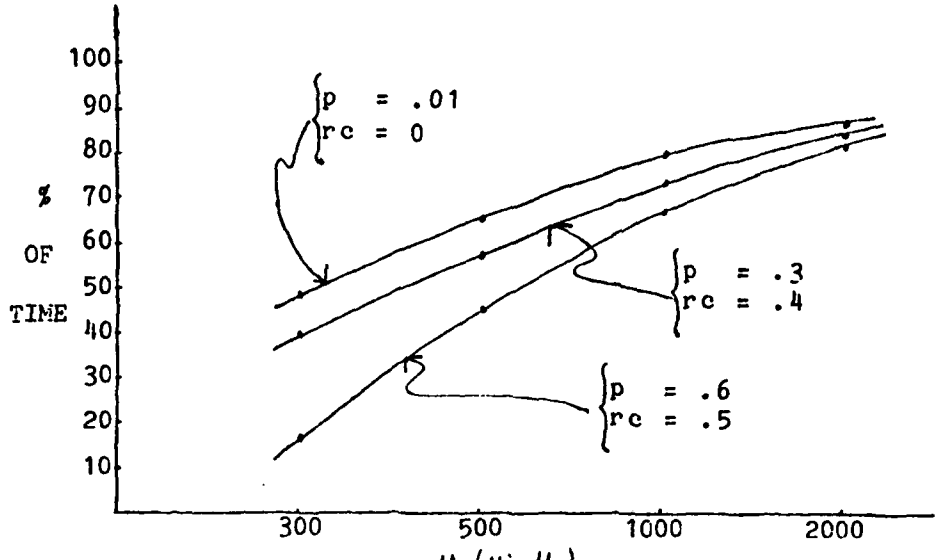


Two Processor Model

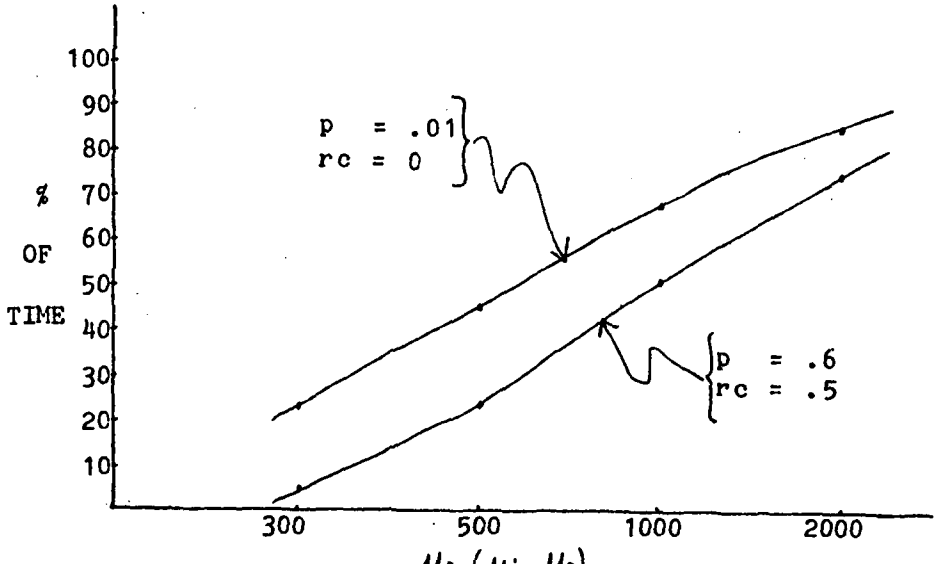


Three Processor Model

Figure 6-6



Two Processor Model



Three Processor Model

Figure 6-7

shown that for a particular algorithm, BSORT, parallelism is increased when another processor is added. This is because the interarrival time of non-local messages increased when the other processor was added. The interarrival time of messages depends on two factors, the process structure and the assignment of processes to processors. Therefore, the way to achieve the maximum amount of parallelism for a given algorithm and hardware configuration, is to find the optimum assignment of processes to processors.

Changes to the parameters μ_0, μ_i, μ_p, and μ_l will cause changes in the degree of parallelism. The parameter μ_l causes a change in parallelism, but not to such an extent as the other parameters. The parameter λ depends on the structure of the application software and the assignment of processes to CPUs. The other three parameters, μ_0, μ_i, and μ_p depend upon the message lengths and the communication bandwidth. It is possible to increase the communication bandwidth to obtain more parallelism, or, on the software end, the process assignment algorithms could be studied to find one that minimizes the frequency of non-local messages.

CHAPTER 7

CONCLUSIONS

7.1 SUMMARY

This thesis performed two tasks: the formulation of a model of parallelism for a distributed computer system, and the instrumentation of that system to validate the model.

The model of parallelism is a Markov process, where the states of the model correspond to the states of the system. A model exists for two and three processors. The two processor model has four states and the three processor model has eight states. It has been determined that a four processor model would have nineteen states and a five processor model, twenty-nine states. These models were not formulated due to the large number of states and the results of validating the two and three processor models indicated that not much information could be gained from them.

Validation of the two and three processor models revealed that they weren't very accurate. The relative error was 20% and less for two processors and 30% and less (with the exception of one state whose error was 55%) for three processors. Composite states, however, exhibit small relative errors. The primary reason for

the discrepencies between the models' output and the measured values was that the models are not able to capture the structure and all of the implications of scheduling, of processes. The model does represent the interprocessor communication patterns. Thus, the structure of the application software plays a significant role in determining the performance of distributed systems.

Instrumenting the XDCS system presented some problems. The measurements of the parameters for the model was straightforward, but the data obtained from sampling the system state was not. It was found that it is mandatory for the system to start exactly the same way every time in order for the sampling data to be consistent. This required a semi-automated startup procedure for the local network instead of a manual one. Also, it is very important to sample at rates whose sampling period is at least twice as fast as the duration of the shortest state. It was found that the sampling rates necessary for this system were very high which prompted the installment of special serial interfaces so that the measurements did not interfere with the execution of the XDCS system.

Information obtained from the model shows that an important factor which influences the degree of parallelism exhibited by the execution of a program is the

assignment of processes to processors. Thus one way to improve the performance of distributed systems would be to find the process assignment which minimizes the frequency of non-local messages and at the same time maximizes parallel activity.

7.1 FUTURE WORK

Further development of this model would not reveal any additional insight into parallelism. The model shows that additional processors result in increased parallelism due to the decrease in the frequency of non-local messages. More information about parallelism could be gained by modeling process structure and characteristics, and the interactions between processes.

Future work in analyzing the performance of distributed computer systems falls into two areas: the application of modeling techniques to support comparisons of decentralized software structures to centralized versions, and the application of modeling techniques to compare alternative, concurrent program designs.

Modeling techniques to support comparisons of decentralized software structures to centralized versions will make it possible to determine the cost of decentralization. At this time, the decentralization cost is substantial; the primary advantages of distributed computer systems are their increased reliability and incre-

mental system growth. However, once the cost of decentralization is quantified and the factors contributing to the cost understood, it might be possible to reduce the cost.

Modeling techniques to analyze alternative program designs are necessary in order to provide a system which performs optimally. These techniques could also indicate how the software should be constructed so that it is compatible with the hardware architecture, or conversely, what hardware architecture would support the software model used. Since the introduction of multiprocessor architectures, compatibility of software and hardware structures has been an important consideration.

BIBLIOGRAPHY

[1] Anderson, G.A., and E.D. Jensen, "Computer Interconnection Structures: Taxonomy, Characteristics and Examples", ACM Computing Surveys, 7, 4, (Dec. 1975), pp. 197-213.

[2] Baer, J.L., "Theoretical Aspects of Multiprocessing" ACM Computing Surveys, 5, 1, (Mar. 1973), pp. 31-80.

[3] Balkovich, E.E., and C. Whitby-Strevens, "On the Performance of Decentralized Software", Proceedings of Performance 80, (May 1986), pp. 173-180.

[4] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, 21, 11, (Nov. 1978), pp. 934-941.

[5] Browne, J.C., K.M. Chandy, J. Hogarth, C.C.A. Lee, "Effect on Throughput of Multiprocessing in a Multiprogramming Environment", IEEE Trans. Computers, Vol. C-22, No. 8, (Aug. 1973), pp. 728-735

[6] Chang, E., and R. Roberts, "An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processors", Communications of the ACM, Vol. 22, No. 5, (April 1979), pp. 281-283

[7] Chang, E., "Decentralized Algorithms in Distributed Systems", Technical Report CSRG-103, University of Toronto, (Oct. 1979).

[8] Cottam, I.D., Functional Specification of the Modula Compiler Release 2, Technical Report, University of York, Department of Computer Science, (March 1979).

[9] Digital Equipment Corporation, Memories and Peripherals, (1980).

[10] Dijkstra, E.W., "Self-Stabilizing Systems in Spite of Distributed Control", Communications of the ACM, Vol. 17, No. 11, (Nov. 1974), pp. 643-644.

[11] Enslow, P.H., Jr., "What is a 'Distributed' Data Processing System?", Computer, 11, 1, (Jan. 1978), pp. 13-21.

[12] Ferrari, D., Computer Systems Performance Evaluation, Prentice-Hall, Englewood Cliffs, NJ, (1978).

[13] Fontaine, S.C., A Distributed Operating System Kernel, M.S. Thesis, Department of Electrical Engineering and Computer Science, University of Connecticut, Storrs, CT, (1980).

[14] Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, 21, 8, (Aug. 1978), pp. 666-677.

[15] Jones, A.K., et. al., "StarOS, A Multiprocessor Operating System for the Support of Task Forces", Proceedings of the Seventh Symposium on Operating System Principles, 10-12 (Dec. 1979), Pacific Grove, CA, pp. 117-127.

[16] Kleinrock, L., Queueing Systems - Volume 1: Theory, John Wiley and Sons, (1975).

[17] Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, 21, 7, (July 1978), pp. 558-565.

[18] LeLann, G., "Distributed Systems - Towards a Formal Approach", 1977 IFIP Congress Proceedings, (1977), pp. 155-160.

[19] LeLann, G., "An Analysis of Different Approaches to Distributed Computing", Proceedings of the 1st International Conference on Distributed Computer Systems, Huntsville, AL, (Oct. 1-5, 1979), pp. 222-232.

[20] Liskov, B., "Primitives for Distributed Computing", Proceedings of the 7-th Symposium on Operating System Principles, Pacific Grove, CA, (Dec. 1979), pp. 33-42.

[21] May, M.D., R.J.B. Taylor, and C. Whitby-Strevens, "EPL - An Experimental Programming Language", IEEE Conference on Trends and Applications: Distributed Computing, Gaithersburg, MD, (May 1978), pp. 69-71.

[22] May, M.D., and R.J.B. Taylor, The EPL Programming Manual, Distributed Computing Report No. 1, Department of Computer Science, University of Warwick, Coventry, England (1979).

[23] Metcalfe, R.M., and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", Communications of the ACM, Vol. 19, No. 7, (July 1976), pp. 395.

[24] Rose, C.A., "A Measurement Procedure for Queueing Network Models of Computer Systems", Computing Surveys, Vol. 10, No. 3, (Sept. 1978), pp. 263-280.

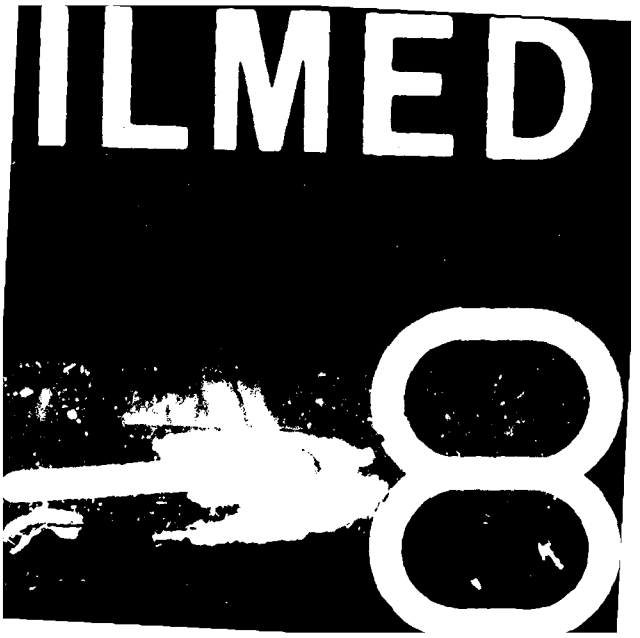
[25] Shoch, J.F., and J.A. Hupp, "Measured Performance of an Ethernet Local Network", Communications of the ACM, 23,12, (Dec. 1980), pp. 711-720.

[26] Solomom, M.H., and R.A. Finkel, "The Roscoe Distributed Operating System", Proceedings of the Seventh Symposium on Operating System Principles, 10-12 (Dec. 1979), Pacific Grove, CA, pp. 108-114.

[27] Stutzman, B.W., "Data Communication Control Procedures", Computing Surveys, Vol. 4, No. 4, (Dec. 1972), pp. 197-220.

[28] Trivedi, K.S., "On the Design and Use of High Performance Computer Systems", Parallel Computers-Parallel Mathematics, Feilmeier (ed.), North Holland, Amsterdam, (1977), pp. 287-291

[29] Whitby-Strevens, C., "Towards the performance evaluation of distributed computing systems", IEEE COMPSAC, Chicago, (1978), pp. 141-146


ILMED
8